3.2 Example Problems The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between standardized and real-world problems. A standardized problem is intended to illustrate or exercise various problem solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms. A real-world problem, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data. Standardized problem Real-world problem 3.2.1 Standardized problems Grid world A grid world problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell. Typically the agent can move to any obstacle-free adjacent cell— horizontally or vertically and in some problems diagonally. Cells can contain objects, which the agent can pick up, push, or otherwise act upon; a wall or other impassible obstacle in a cell prevents an agent from moving into that cell. The vacuum world from Section 2.1 can be formulated as a grid world problem as follows: STATES: A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. In the simple two-cell version, the agent can be in either of the two cells, and each call can either contain dirt or not, so there are states (see Figure 3.2). In general, a vacuum environment with cells has states. Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: INITIAL STATE: Any state can be designated as the initial state. ACTIONS: In the two-cell world we defined three actions: Suck, move Left, and move Right. In a two-dimensional multi-cell world we need more movement actions. We could add Upward and Downward, giving us four absolute movement actions, or we could switch to egocentric actions, defined relative to the viewpoint of the agent—for example, Forward, Backward, TurnRight, and TurnLeft. TRANSITION MODEL: Suck removes any dirt from the agent's cell; Forward moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect. Backward moves the agent in the opposite direction, while TurnRight and TurnLeft change the direction it is facing by GOAL STATES: The states in which every cell is clean. ACTION COST: Each action costs 1. $2 \cdot 2 \cdot 2 = 8$ n n $\cdot$ 2 n L = Left, R = Right, S = Suck. 90 ∘ .Sokoban puzzle Another type of grid world is the sokoban puzzle, in which the agent's goal is to push a number of boxes, scattered about the grid, to designated storage locations. There can be at most one box per cell. When an agent moves forward into a cell containing a box and there is an empty cell on the other side of the box, then both the box and the agent move forward. The agent can't push a box into another box or a wall. For a world with non-obstacle cells and boxes, there are states; for example on an grid with a dozen boxes, there are over 200 trillion states. In a sliding-tile puzzle, a number of tiles (sometimes called blocks or pieces) are arranged in a grid with one or more blank spaces so that some of the tiles can slide into the blank space. One variant is the Rush Hour puzzle, in which cars and trucks slide around a grid in an attempt to free a car from the traffic jam. Perhaps the best-known variant is the 8- puzzle (see Figure 3.3), which consists of a grid with eight numbered tiles and one blank space, and the 15-puzzle on a grid. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation of the 8 puzzle is as follows: STATES: A state description specifies the location of each of the tiles. INITIAL STATE: Any state can be designated as the initial state. Note that a parity property partitions the state space—any given goal can be reached from exactly half of the possible initial states (see Exercise 3.PART). ACTIONS: While in the physical world it is a tile that slides, the simplest way of describing an action is to think of the blank space moving Left, Right, Up, or Down. If the blank is at an edge or corner then not all actions will be applicable. TRANSITION MODEL: Maps a state and action to a resulting state; for example, if we apply Left to the start state in Figure 3.3 , the resulting state has the 5 and the blank switched. Figure 3.3 n b n × n!/(b!(n − b)!) 8 × 8 6 × 6 3 × 3 4 × 4 A typical instance of the 8-puzzle. GOAL STATE: Although any state could be the goal, we typically specify a

state with the numbers in order, as in Figure 3.3 . ACTION COST: Each action costs 1. Sliding-tile puzzle 8-puzzle 15-puzzle Note that every problem formulation involves abstractions. The 8-puzzle actions are abstracted to their beginning and final states, ignoring the intermediate locations where the tile is sliding. We have abstracted away actions such as shaking the board when tiles get stuck and ruled out extracting the tiles with a knife and putting them back again. We are left with a description of the rules, avoiding all the details of physical manipulations. Our final standardized problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise. Knuth conjectured that starting with the number 4, a sequence of square root, floor, and factorial operations can reach any desired positive integer. For example, we can reach 5 from 4 as follows: The problem definition is simple: STATES: Positive real numbers. INITIAL STATE: 4. ACTIONS: Apply square root, floor, or factorial operation (factorial for integers only). TRANSITION MODEL: As given by the mathematical definitions of the operations. GOAL STATE: The desired positive integer. ACTION COST: Each action costs 1. The state space for this problem is infinite: for any integer greater than 2 the factorial operator will always yield a larger integer. The problem is interesting because it explores very large numbers: the shortest path to 5 goes through Infinite state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects. 3.2.2 Real-world problems We have already seen how the route-finding problem is defined in terms of specified locations and transitions along edges between them. Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example. (The main complications are varying costs due to traffic-dependent delays, and rerouting due to road closures.) Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications. Consider the airline travel problems that must be solved by a travel-planning Web site: STATES: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous ⌊ √ √ √√(4!)!⌋ = 5. (4!)! = 620,448,401,733,239,439,360,000.segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects. INITIAL STATE: The user’s home airport. ACTIONS: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed. TRANSITION MODEL: The state resulting from taking a flight will have the flight’s destination as the new location and the flight’s arrival time as the new time. GOAL STATE: A destination city. Sometimes the goal can be more complex, such as “arrive at the destination on a nonstop flight.” ACTION COST: A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on. Commercial travel advice systems use a problem formulation of this kind, with many additional complications to handle the airlines’ byzantine fare structures. Any seasoned traveler knows, however, that not all air travel goes according to plan. A really good system should include contingency plans—what happens if this flight is delayed and the connection is missed? Touring problems describe a set of locations that must be visited, rather than a single goal destination. The traveling salesperson problem (TSP) is a touring problem in which every city on a map must be visited. The aim is to find a tour with cost (or in the optimization version, to find a tour with the lowest cost possible). An enormous amount of effort has been expended to improve the capabilities of TSP algorithms. The algorithms can also be extended to handle fleets of vehicles. For example, a search and optimization algorithm for routing school buses in Boston saved $5 million, cut traffic and air pollution, and saved time for drivers and students (Bertsimas et al., 2019). In addition to planning trips, search algorithms have been used for tasks such as planning the movements of automatic circuit board drills and of stocking machines on shop floors. Touring problem < CTraveling salesperson problem (TSP) A VLSI layout problem requires

positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually split into two parts: cell layout and channel routing. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving. VLSI layout Robot navigation is a generalization of the route-finding problem described earlier. Rather than following distinct paths (such as the roads in Romania), a robot can roam around, in effect making its own paths. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs that must also be controlled, the search space becomes many-dimensional—one dimension for each joint angle. Advanced techniques are required just to make the essentially continuous search space finite (see Chapter 26). In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls, with partial observability, and with other agents that might alter the environment. Robot navigation Automatic assembly sequencing of complex objects (such as electric motors) by a robot has been standard industry practice since the 1970s. Algorithms first find a feasible assembly sequence and then work to optimize the process. Minimizing the amount of manual human labor on the assembly line can produce significant savings in time and cost. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking an action in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space. One important assembly problem is protein design, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease. Automatic assembly sequencing Protein design3.3 Search Algorithms A search algorithm takes a search problem as input and returns a solution, or an indication of failure. In this chapter we consider algorithms that superimpose a search tree over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state. Each node in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem. Search algorithm Node It is important to understand the distinction between the state space and the search tree. The state space describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another. The search tree describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees). Expand Figure 3.4 shows the first few steps in finding a path from Arad to Bucharest. The root node of the search tree is at the initial state, Arad. We can expand the node, by considering the available ACTIONS for that state, using the RESULT function to see where those actions lead to, and generating a new node (called a child node or successor node) for each of the resulting states. Each child node has Arad as its parent node. Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are lavender with bold letters; nodes on the frontier that have been generated but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been reached. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad

to Sibiu to Arad; that can't be an optimal path, so search should not continue from there. GeneratingChild node Successor node Parent node Now we must choose which of these three child nodes to consider next. This is the essence of search—following up one option now and putting the others aside for later. Suppose we choose to expand Sibiu first. Figure 3.4 (bottom) shows the result: a set of 6 unexpanded nodes (outlined in bold). We call this the frontier of the search tree. We say that any state that has had a node generated for it has been reached (whether or not that node has been expanded). Figure 3.5 shows the search tree superimposed on the state-space graph. 5 Some authors call the frontier the open list, which is both geographically less evocative and computationally less appropriate, because a queue is more efficient than a list here. Those authors use the term closed list to refer to the set of previously expanded nodes, which in our terminology would be the reached nodes minus the frontier. Figure 3.5 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1 . At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea. □ 5 □ □Frontier Reached Note that the frontier separates two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached. This property is illustrated in Figure 3.6 . Figure 3.6 The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order. Separator 3.3.1 Best-first search How do we decide which node from the frontier to expand next? A very general approach is called best-first search, in which we choose a node, with minimum value of some □ n,evaluation function, Figure 3.7 shows the algorithm. On each iteration we choose a node on the frontier with minimum value, return it if its state is a goal state, and otherwise apply EXPAND to generate child nodes. Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path. The algorithm returns either an indication of failure, or a node that represents a path to a goal. By employing different functions, we get different specific algorithms, which this chapter will cover. Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2 . See Appendix B for yield. Best-first search Evaluation function f(n). □ f(n) f(n) □ □3.3.2 Search data structures Search algorithms require a data structure to keep track of the search tree. A node in the tree is represented by a data structure with four components: node.STATE: the state to which the node corresponds; node.PARENT: the node in the tree that generated this node; node.ACTION: the action that was applied to the parent's state to generate this node; node.PATH-COST: the total cost of the path from the initial state to this node. In mathematical formulas, we use as a synonym for PATH-COST. Following the PARENT pointers back from a node allows us to recover the states and actions along the path to that node. Doing this from a goal node gives us the solution. We need a data structure to store the frontier. The appropriate choice is a queue of some kind, because the operations on a frontier are: IS-EMPTY(frontier) returns true only if there are no nodes in the frontier. POP(frontier) removes the top node from the frontier and returns it. TOP(frontier) returns (but does not remove) the top node of the frontier. ADD(node, frontier) inserts node into its proper place in the queue. Queue Three kinds of queues are used in search algorithms: A priority queue first pops the node with the minimum cost according to some evaluation function, It is used in best-first search. g(node) f.Priority queue A FIFO queue or first-in-

first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search. FIFO queue A LIFO queue or last-in-first-out queue (also known as a stack) pops first the most recently added node; we shall see it is used in depth-first search. LIFO queue Stack The reached states can be stored as a lookup table (e.g. a hash table) where each key is a state and each value is the node for that state. 3.3.3 Redundant paths The search tree shown in Figure 3.4 (bottom) includes a path from Arad to Sibiu and back to Arad again. We say that Arad is a repeated state in the search tree, generated in this case by a cycle (also known as a loopy path). So even though the state space has only 20 states, the complete search tree is infinite because there is no limit to how often one can traverse a loop. Repeated state Cycle Loopy path A cycle is a special case of a redundant path. For example, we can get to Sibiu via the path Arad–Sibiu (140 miles long) or the path Arad–Zerind–Oradea–Sibiu (297 miles long). This second path is redundant—it's just a worse way to get to the same state—and need not be considered in our quest for optimal paths. Redundant path Consider an agent in a grid world, with the ability to move to any of 8 adjacent squares. If there are no obstacles, the agent can reach any of the 100 squares in 9 moves or fewer. But the number of paths of length 9 is almost (a bit less because of the edges of the grid), or more than 100 million. In other words, the average cell can be reached by over a million redundant paths of length 9, and if we eliminate redundant paths, we can complete a search roughly a million times faster. As the saying goes, algorithms that cannot remember the past are doomed to repeat it. There are three approaches to this issue. First, we can remember all previously reached states (as best-first search does), allowing us to detect all redundant paths, and keep only the best path to each state. This is appropriate for state spaces where there are many redundant paths, and is the preferred choice when the table of reached states will fit in memory. 10 × 10 8 9Graph search Tree-like search Second, we can not worry about repeating the past. There are some problem formulations where it is rare or impossible for two paths to reach the same state. An example would be an assembly problem where each action adds a part to an evolving assemblage, and there is an ordering of parts so that it is possible to add and then but not and then For those problems, we could save memory space if we don't track reached states and we don't check for redundant paths. We call a search algorithm a graph search if it checks for redundant paths and a tree-like search if it does not check. The BEST-FIRST-SEARCH algorithm in Figure 3.7 is a graph search algorithm; if we remove all references to reached we get a tree-like search that uses less memory but will examine redundant paths to the same state, and thus will run slower. 6 We say "tree-like search" because the state space is still the same graph no matter how we search it; we are just choosing to treat it as if it were a tree, with only one path from each node back to the root. Third, we can compromise and check for cycles, but not for redundant paths in general. Since each node has a chain of parent pointers, we can check for cycles with no need for additional memory by following up the chain of parents to see if the state at the end of the path has appeared earlier in the path. Some implementations follow this chain all the way up, and thus eliminate all cycles; other implementations follow only a few links (e.g., to the parent, grandparent, and great-grandparent), and thus take only a constant amount of time, while eliminating all short cycles (and relying on other mechanisms to deal with long cycles). 3.3.4 Measuring problem-solving performance Before we get into the design of various search algorithms, we will consider the criteria used to choose among them. We can evaluate an algorithm's performance in four ways: A B, B A. 6 COMPLETENESS: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not? Completeness COST OPTIMALITY: Does it find a solution with the lowest path cost of all solutions? 7 Some authors use the term "admissibility" for the property of finding the lowest-cost solution, and some use just "optimality," but that can be confused with other types of optimality. Cost optimality TIME COMPLEXITY: How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered. Time complexity SPACE COMPLEXITY:

How much memory is needed to perform the search? Space complexity To understand completeness, consider a search problem with a single goal. That goal could be anywhere in the state space; therefore a complete algorithm must be capable of systematically exploring every state that is reachable from the initial state. In finite state 7spaces that is straightforward to achieve: as long as we keep track of paths and cut off ones that are cycles (e.g. Arad to Sibiu to Arad), eventually we will reach every reachable state. In infinite state spaces, more care is necessary. For example, an algorithm that repeatedly applied the “factorial” operator in Knuth’s “4” problem would follow an infinite path from 4 to 4! to (4!)!, and so on. Similarly, on an infinite grid with no obstacles, repeatedly moving forward in a straight line also follows an infinite path of new states. In both cases the algorithm never returns to a state it has reached before, but is incomplete because wide expanses of the state space are never reached. To be complete, a search algorithm must be systematic in the way it explores an infinite state space, making sure it can eventually reach any state that is connected to the initial state. For example, on the infinite grid, one kind of systematic search is a spiral path that covers all the cells that are steps from the origin before moving out to cells that are steps away. Unfortunately, in an infinite state space with no solution, a sound algorithm needs to keep searching forever; it can’t terminate because it can’t know if the next state will be a goal. Systematic Time and space complexity are considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state-space graph, where is the number of vertices (state nodes) of the graph and is the number of edges (distinct state/action pairs). This is appropriate when the graph is an explicit data structure, such as the map of Romania. But in many AI problems, the graph is represented only implicitly by the initial state, actions, and transition model. For an implicit state space, complexity can be measured in terms of the depth or number of actions in an optimal solution; the maximum number of actions in any path; and the branching factor or number of successors of a node that need to be considered. Depth s s + 1 |V | + |E|, |V | |E| d, m, b,Branching factor.